LexiDB: Patterns & Methods for Corpus Linguistic Database Management

Matthew Coole, Paul Rayson, John Mariani
Lancaster University
Lancaster, UK
{m.coole, p.rayson, j.mariani}@lancaster.ac.uk

Abstract

LexiDB is a tool for storing, managing and querying corpus data. In contrast to other database management systems (DBMSs), it is designed specifically for text corpora. It improves on other corpus management systems (CMSs) because data can be added and deleted from corpora on the fly with the ability to add live data to existing corpora. LexiDB sits between these two categories of DBMSs and CMSs, more specialised to language data than a general-purpose DBMS but more flexible than a traditional static corpus management system. Previous work has demonstrated the scalability of LexiDB in response to the growing need to be able to scale out for ever-growing corpus datasets. Here, we present the patterns and methods developed in LexiDB for storage, retrieval and querying of multi-level annotated corpus data. These techniques are evaluated and compared to an existing CMS (Corpus Workbench CWB - CQP) and indexer (Lucene). We find that LexiDB consistently outperforms existing tools for corpus queries. This is particularly apparent with large corpora and when handling queries with large result sets.

Keywords: corpus, database

1. Introduction

Recent years have seen a development of many new innovative DBMSs (Database Management Systems) that have moved away from the traditional relational database model. The No-SQL movement, in particular, has concerned itself largely with a system's ability to scale out to handle increasingly large datasets. Language data in the worlds of corpus and computational linguistics, digital humanities and lexicography has also seen the scale of what is considered a 'large' corpus grow from one million words in LOB (Johansson et al., 1986), and one hundred million words British National Corpus (Leech, 1992) to billions and tens of billions of words TenTen (Jakubíček et al., 2013), COW (Schäfer and Bildhauer, 2012), Hansard[1], EEBO-TCP[2] over the last forty years. However, specialist corpus storage and retrieval tools have lagged behind the developments made by more generic DBMSs. In addition, modern DBMSs do not provide the necessary query functionality to satisfy the types of queries used by corpus linguists. Moreover, existing database models do not provide efficient storage and retrieval for consecutive words in running text, sequential queries and contextual sorting that are central to the requirements for textual corpora. These issues present a problem for linguists, humanists and others wishing to work with extreme scale corpus datasets using the corpus methodology.

Conventional solutions to storage and querying of corpus datasets (such as relational DBMSs and index tools such as Lucene[3]) have worked in the past for smaller-scale datasets. However, they have always relied upon an application layer built on top of a DBMS to translate specialist corpus queries into a query for a more generic data model such as Brigham Young's system (Davies, 2005). As a result, many corpus linguistic searches require multiple queries to the underlying data. With smaller datasets this is feasible but with billions of records, such practises fail to provide responsive systems. Clearly, the underlying data model and indexing system used needs to be tailored to provide the ability to perform linguistic queries in a scalable way.

The novel database design and system that we propose, LexiDB, provides a means of storing and querying extreme-scale corpus data that can not only fulfil specialist linguistic queries but can also be scaled out effectively to handle extremely large corpus datasets consisting of billions of words. Unlike existing solutions, its data model is tailored for text corpora and as such can provide a better means of access and partitioning of data to allow for reduced parsing time (time taken to insert and index a whole corpus) and the ability to dynamically scale-out. LexiDB uses a specialised text storage system which maps tokens to integer values. This allows for text to be stored in a way that allows for faster index lookups and faster lexical sorting of query results.

Previous work (Coole et al., 2015) in this area has demonstrated that existing DBMSs do not support the ability to handle extreme-scale corpora. This work demonstrated the difficulty that both Cassandra and MongoDB had in fulfilling simple concordance line queries without multiple database queries. It also illustrated issues with data insertions with no ability to perform a truly parallel insertion even with a clustered setup. This means insertion or parsing of extremely large corpora of the order of multiple billions of words such as UK Hansard on a four-machine cluster takes over eight hours.

Although this previous work demonstrated existing DBMSs ability to scale up to higher data volumes, it also showed their inability to support both text indexing and data access tailored for corpus queries. Furthermore, existing DBMSs were shown to lack the degree of parallelism required for parsing in extreme-scale corpus datasets en mass with a design philosophy that appears more geared towards building a large database over time.

2. Related Work

The language stored in text corpora can be analysed using various techniques. Here, we focus on the methodology

[1] http://www.gla.ac.uk/schools/critical/research/fundedresearchprojects/parliamentarydiscourse/
[2] http://www.textcreationpartnership.org/tcp-eebo/
[3] https://lucene.apache.org/

used mainly in corpus linguistics (McEnery and Hardie, 2012), digital humanities (Gregory and Paterson, 2019), and to some extent lexicography (Corréard, 2002).

A common technique used in these fields is concordance analysis. This focuses on a particular keyword or phrase and examines the surrounding patterns of language to learn about a word's meaning or grammatical patterning. Concordance lines are built from the words that occur immediately adjacent to the keyword, thus giving the linguist an idea of the typical context in which a keyword is used. This is more commonly known outside of linguistics as a KWIC (keyword in context) search. A linguist can draw conclusions about patterns found around the queried term such as in a study conducted on the portrayal of Muslims in the British Press (Baker et al., 2012). The validity of such conclusions regarding the use of language is of course predicated upon having a sufficient number of concordance lines to examine and a representative corpus. Very common words can be examined in relatively small corpora. For less frequent words or phrases, it may be necessary to build larger and larger corpora to gather sufficient data for a meaningful concordance analysis. Along with the rise in availability of web-derived corpus data, this has led to the ever-increasing size of corpora and an ever greater need for tools and systems capable of handling them.

Collocations are another feature in corpora that are of interest to linguists. These are potential discontinuous sequences of two or more words that occur together with higher frequency. Thus they may represent a significant linguistic feature or pattern in a language variety. N-grams are also used to search for patterns alongside collocations. These are usually a consecutive sequence of a given number (n) of words that occur frequently together in a corpus. A complete set of bi-grams (2-grams) for a corpus would contain every pair of words that occur next to each other. Storing n-grams along with their frequency is typical. This allows linguists to examine how likely they are to occur and to study their variation across different genres or text types. No more than five-word patterns (5-grams) are usually stored. Browsing or searching of n-grams allows linguists to look for many different types of lexical, grammatical or semantic features of interest.

Various tools exist which implement the five main methods in corpus linguistics: concordances, collocation, frequency lists, keywords and n-grams. AntConc (Anthony, 2004) is one of the most popular tools to query corpora in this way as it allows users to do so from the raw files. AntConc struggles to process larger corpora (i.e. billions of words), as does Wmatrix (Rayson, 2008), a web-based corpus annotation and comparison tool. IntelliText[4] provides a web interface to a selection of pre-loaded corpora that have already been indexed. The system allows access to some large corpora such as the British National Corpus (BNC). This system suffers similar pitfalls (slow retrieval and/or inability to retrieve) when searching large corpora for particularly common words but is very efficient in the general case; to remedy this, limits on the number of results returned are applied.

Brigham Young University's system (BYU)[5] provides an interface built on a relational database (Davies, 2005) that gives access to several extreme-scale corpora such as the corpus of Global Web-Based English (GloWbE) and the Wikipedia corpus (both 1.9 billion words in size). BYU's interface provides a means to generate concordance lines or KWICs as well as a means to sort them in several meaningful ways such as lexically on one word left of the search term. As with IntelliText, BYU also limits the number of returned results to avoid the issues associated with huge result sets in extremely large corpora – a common pattern with corpus systems.

Systems based on the IMS Open Corpus Workbench such as CQPweb (Hardie, 2012) currently have a hard upper limit (2.1 billion tokens) on the size of corpora that they can index. Other systems, such as Corpuscle[6] are targetted at corpora of the order of one billion words. SketchEngine and the open-source cut-down version incorporating Manatee (corpus management tool) and Bonito (graphical interface) (Rychlý, 2007) has an apparent maximum allowed corpus size of $2^{31} - 2$ according to the Manatee change log but no benchmarks are available for indexing or retrieval speed.

Varying different requirements of corpus linguists and the various tools, both general-purpose and bespoke, has led to some corpus management platforms such as Korap (Diewald et al., 2016) being developed. Korap contains various components including a front-end and API interface, management component and a back end data system with an indexing structure built on Lucene. Whilst this type of structure may be daunting for casual users to get to grips with Korap does address a key issue in being able to use its Koral component to translate queries in various corpus query languages such as CQL (Evert, 2005) and Annis QL (Chiarcos et al., 2008) into its own query language for execution. This provides a means of easy entry for linguists familiar with supported query languages.

Many different approaches have been taken to attempt to resolve the problems associated with extreme-scale corpora and the difficulty in fulfilling corpus queries that they present. The Ziggurat (Evert and Hardie, 2015) design attempts to resolve issues relating to multi-layered corpora and the more complex queries that may be employed by linguists with such corpora, but distribution is not part of the design. Chubak et al (Chubak and Rafiei, 2010) attempt to resolve issues of scalability by storing more data within the index i.e. context data about surrounding terms building on the work of Cafarella (Cafarella and Etzioni, 2005). This approach could be extended but requires additional data to be stored in the index, which will always be a limiting factor. Schneider (Schneider, 2012) proposes linking an object-relational DBMS to a MapReduce style system with a 4-billion-word test corpus. However, his experiments are only carried out on a single computer with single or multi-processor options, rather than fully distributed. Several techniques have been applied towards satisfying the data needs of large corpora but few have managed to address the fundamental scalability problem.

[4] http://corpus.leeds.ac.uk/

[5] http://corpus.byu.edu/

[6] http://uni.no/en/uni-computing/clu/corpuscle/

3. System

LexiDB is designed to be a distributed corpus database; the core principle is to promote parallelism in the insertion and querying of corpus data. The system makes use of parallel programming techniques to facilitate the ability to scale out across an ever-increasing number of nodes as the volume of data in the corpus scales up.

Many corpus systems currently exist that are as functionally capable as LexiDB's prototype, e.g. with an existing user-interface. However, none are known to be capable of scaling out in the same manner. Furthermore, it is not the intention of LexiDB to replace these systems or to duplicate their functionality directly but rather to provide a potential data layer upon which future corpus tools and software can be built allowing them to be scalable for extremely large corpora.

The distribution model used by LexiDB is peer-to-peer (p2p) with no one node being a single point of failure. This architecture is appropriate for the currently intended deployment environments i.e. less than 10 nodes. For larger deployment more sophisticated network topologies based on p2p principles of decentralization (utilised by Gnutella (Frankel and Pepper, 1999) and Cassandra[7] to some extent) or other techniques such as DHTs (Distributed Hash Tables), may be used to minimise network traffic and to allow for even greater scalability of the design. The current design ensures all nodes within the network are aware of all other nodes and can distribute queries accordingly. As a result certain queries may be redundantly run on nodes with no data pertaining to them, but given the expense of most queries to run on each node this will have very little effect on the responsiveness of the system.

The distribution aspect of LexiDB has been previously discussed in greater depth and evaluated in previous work (Coole et al., 2016). This demonstrated how well corpus data can be scaled out with consistent returns on efficiency for both inserting and querying data as the number of nodes in the cluster grows. What follows is a discussion of the underlying data storage patterns and querying methods that are used by LexiDB and how the system as a whole fits together, allowing it to handle language data and corpus queries. LexiDB can support both annotated corpus layers and persistent metadata. Metadata can be anything but typical examples are; author, publication date, origin etc.

3.1. Data Block Partitioning

To maintain a greater level of flexibility and manageability, data is broken down within each node into data blocks. Each data block contains, as a default, 1 million words or records. The size of each block can be tailored, and increasing the limit may allow for faster querying but may slow parse time. All blocks are independently managed and as a result, can easily be migrated between nodes allowing for additional nodes to be added and the system to quickly redistribute the data without the need to fully rebuild any blocks or block-level indexes. Breaking the data down in this way allows the initial parse of data in a corpus to be performed far quicker than processing the data as a single chunk.

[7] http://cassandra.apache.org/

3.2. Numeric Data Representation

The basic principle of how data within LexiDB is stored is that all words are converted to a numeric representation. The numeric representations should be ordered alphabetically according to the word they represent. Each word type will share the same numeric representation. Each word type from the phrase is placed into an alphabetically ordered dictionary. Once all words are in the dictionary, they are assigned ascending numeric values. The original values of the phrase are replaced by the values from the dictionary and stored. For quick reverse retrieval when performing queries this dictionary is also stored. Storing all words in this way allows for several advantages including the ability to sort large sets of words far quicker as well as affording a small level of compression.

Since each block is computed separately at parse time, each block has a unique numeric representation for particular words. This presents a problem when dealing with data that is retrieved across different blocks. Once all blocks of a corpus have been computed, their dictionaries become the input files for the same process i.e. all dictionaries are taken and numeric representations of the words they contain are created and stored at a node level. A mapping file is then constructed for each block which will map its block-level numeric representation of a word to a node-level numeric representation of a word.

3.3. Index Structure

LexiDB uses a block-level indexing structure. Within each block, all stored numeric values are indexed. The index file is compressed using PFOR-delta[8] scheme as a postings list of the positions where each value occurs. The postings list for each numeric value are stored contiguously, to look up values a second index map file is created that stores the position within the main index file where the postings list for each numeric value begins. Once the numeric value for the word being queried is retrieved the index looks up that value in the index map which gives the position that the index entries for that value begin in the main index file (the number of index entries is computed by checking where the next set of values begin i.e. checking the next value in the index map). From here all index entries can be retrieved from the main index file and then the values can be retrieved. Typically when retrieving a record a context is used so for each index entry n records before and after will also be retrieved.

3.4. Distribution

LexiDB uses a distributed hash table (DHT) to allow it to be deployed across multiple nodes. The hash for each data block is computed from the ingested data itself. Previous work (Coole et al., 2016) has demonstrated the scalability of this approach. The chord algorithm (Stoica et al., 2003) is the basis for the DHT implementation and is used to evenly distribute data blocks between nodes in a distributed setup. When querying each data block is individually queried with the processing performed by the node where the block is primarily stored and the results alone returned (as opposed

[8] https://github.com/lemire/JavaFastPFOR

to retrieving the entire data block from the DHT and then processing). This method allows for a dynamic cluster configuration to be achieved with the ability to add and remove nodes whilst ensuring a degree of parallelism when querying the system.

3.5. Query syntax

The query syntax of LexiDB combines a basic regular expression syntax and JSON (javascript object notation). JSON objects replace characters within the regular expression syntax and represent QBE (query by example) database objects. Each QBE object contains a series of key-value pairs, where the keys should correspond to column names in the database and the value can be expressed as string values or regular expressions themselves.

```
{"pos":"JJ"}*{"pos":"NN1"}
```

The above example using the CLAWS7[9] tagset values on a POS (part-of-speech) column. This would represent a search for zero or more adjectives (JJ), followed by a singular noun (NN1).

This style of query syntax is similar to CWB/CQP's query language, but it does not seek to extend and/or simplify but rather combine existing languages (regex & JSON) in a meaningful way to create an expressive basis for defining corpus queries.

From a practical standpoint the problem of resolving such queries within a database structure described above can be split into two distinct parts; resolving QBE objects and resolving token stream regular expressions. What follows are two algorithms for solving these problems as well as a discussion of the implementation considerations around how these algorithms work within LexiDB.

3.6. Resolving QBE objects

To resolve token stream regular expressions of the form described above, first QBE objects within this must be resolved. These QBE object will take key-value pairs where the values may be regular expressions themselves to be matched against a dictionary or lexicon of values for that key/database column. The Corpuscle (Meurer, 2012) system made use of suffix arrays to do this. This approach can be taken a stage further when combined with the approach presented by Baeza-Yates (Baeza-Yates and Gonnet, 1996) in resolving against a patricia-trie against binary DFAs (deterministic finite-state automata). Generalizing towards a radix-trie allows for a lexicon to be searched against a regular expression when it is expressed as a DFA. This also allows for a practical approach that can facilitate the use of many existing regex libraries without the need for a bespoke regex engine. LexiDB makes use of the dk.brics DFA library (Møller, 2017).

Preamble: Take any regular expression that can be converted to an equivalent DFA, represented as $M = (Q, \Sigma, \delta, q_0, F)$. Take a dictionary of token types ϵ and compile them into a depth reduced radix trie $T = (r_0 R, E, L)$ where R is the set of all nodes in the trie numbered in a breadth-first traversal

[9] http://ucrel.lancs.ac.uk/claws7tags.html

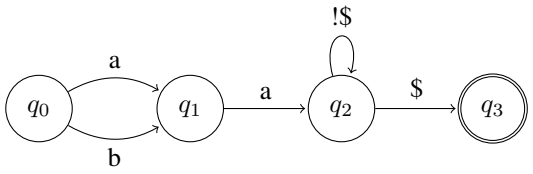

Figure 1: DFA representing `^[ab]a.*$`

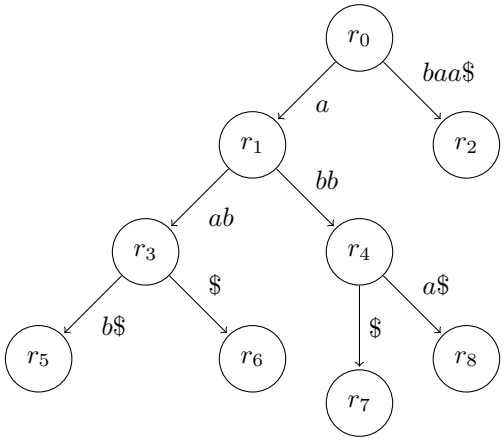

Figure 2: Radix trie representing ϵ

i.e. r_0 is the root node; $r_0, r_1 ... r_n$, E is a set of all edges in the trie expressed as a tuple (r_i, I, r_j) where I is the input sequence from node r_i to r_j and L represents a set of leaf nodes $L \subset R$ and corresponds to the set of token types in the dictionary ϵ.

input : A DFA $M = (Q, \Sigma, \delta, q_0, F)$
Radix trie $T = (r_0, R, E, L)$
output A set of integer pairs h
:

```
h ← ∅
k ← {(q_0, r_0)}
while k ≠ ∅ do
    (q_i, ri) ← k.pop()
    for e in r_i do
        if δ(q_i,e)ϵF then
            δ(q_i,e) ∪ h
        else if δ(q_i,e)ϵQ then
            (δ(q_i,e), R(r_i, e)) ∪ k
    end
end
```

Algorithm 1: Algorithm for resolving a DFA over a radix trie

3.7. Resolving token stream expressions

Once QBE objects within a token stream regular expression have been resolved to specific positions within the corpus it is necessary to resolve the regular expression as a whole to find a set of final positions that match the phrase being searched for. Algorithm 2 attempts to do this in a way that is agnostic of any such regular expression engine, as such as with resolving regular expressions within QBE object LexiDB does not have its own regular expression engine but

uses dk.brics.automaton[10] which can be easily swapped for another library if other regular expression language features become a requirement in the future (the only stipulation on the library used is that it uses DFAs to resolve regular expressions).

Preamble: Take any query Q made up of a set of QBE objects $a_0 ... a_n$ and regular expression operators. Convert these QBE object to pairing of character representation ($a_0 \rightarrow$ 'a', $a_1 \rightarrow$ 'b' etc.) and resolved index lookup positions within a corpus ('a' = 0,12,15...). Map these characters onto their respective index positions in a sparse 2D matrix C representing the entire corpus. Replace the QBE object in the original expression with their character mapped representations and construct a DFA; $M = (Q, \Sigma, \delta, q_0, F)$.

input : A DFA $M = (Q, \Sigma, \delta, q_0, F)$
Sparse 2D matrix C of length l
output A set of integer pairs h
:

$h \leftarrow \emptyset$
for $i \leftarrow 0$ **to** l **do**
  $S \leftarrow [(\mathrm{q}_0, i)]$
  while $S \neq \emptyset$ **do**
    $(q_n, k) \leftarrow$S.$pop()$
    for $j \leftarrow 0$ **to** $C[k].length$ **do**
      if $\delta(q_n, C[k][j])$ **then**
        S.push($\delta(q_n, C[k][j]), k + 1$)
      end
      if $\delta(q_n, C[k][j])$ *in* F **then**
        h.add($(i, k + 1)$)
      end
    end
  end
end

Algorithm 2: Algorithm for resolving QBE based regular expressions over a token stream

Consider the token stream regular expression q_0*q_1 where the index lookup results of resolving the QBE objects are $q_0 = 0, 1, 5, 6$ and $q_1 = 1, 4, 5$ in some corpus of length 7. The token stream regex can be converted to a typical character regex a*b and the sparse 2D matrix representation of this corpus can be built as such `[[a],[a,b],[],[],[b],[a,b],[a]]`. Passing this through algorithm 2 would produce h = `[(0,1),(1,1),(4,4),(5,5)]`.

Further adaptations and performance improvements can be made to this algorithm in practice. The two specific optimizations to consider are a maximum hit length and a greedy matching option (both of which are configurable options in LexiDB).

A greedy match ensures that only the longest possible match is retrieved and no potential sub-sequences that could match on the regular expression are matched. This would typically be seen as a duplicate match by a corpus linguist and as such greedy match is the default option within LexiDB.

A maximum match length is a two faceted performance improvement. On one hand, it allows LexiDB to avoid the potential for a state explosion whilst parsing the DFA and ensures a mechanism is available to bail-out if necessary. Secondly, it facilitates taking advantage of the typical sparsity of the 2D character matrix representing the corpus, meaning the algorithm only needs to execute from positions in the matrix within a proximity of an entry in the matrix, since even if the regular expression begins with wildcards i.e. dot and Kleene star operators, the maximum match length would be exceeded anyway outside this range. Conversely, this means that any token stream regular expression must contain at least one QBE object and not wildcards alone. LexiDB's default maximum match length is 20 tokens.

In practice, this maximum match length can act as a necessary overlap for breaking down the corpus representation into chunks to be parallelised when processed by LexiDB. Since all data is read from the disk when resolving the QBE expressions and the resultant matches are retrieved subsequently when the positions have been found via Algorithm 2 this can scale to as many processors/cores as are available and should reduce the execution time by the same order. This again is another configurable option within LexiDB.

4. Evaluation

4.1. Setup

To evaluate the performance of lexiDB we compared the query response time of searching for the most common uni, bi & trigrams on part-of-speech (C5) tags in the British National Corpus. The performance was compared to the query response time of CWB - CQP. Further to this, the indexing lookup times were compared to the index lookup times of Lucene (since many other search systems and corpus tools use this as an indexer e.g. Korap, ElasticSearch).

All benchmarks were conducted on the same workbench, specs: i7, 16GB RAM, 256GB SSD (271 MB/s read speeds measured by bonni++[11]). The top 20 unigrams, bigrams and trigrams were queried on each of the supported systems 10 times and the average response times are plotted below. In the case of lexiDB, the query cache was disabled so each result would not be retrieved from the cache after each run, rather than performing the query again. The corpus used in all tests was the BNC (British National Corpus) with annotation layers; POS, lemma & C5 tags.

For each of the test systems, the original XML format of the BNC needed to be converted to a format that was supported. LexiDB supports the use of standard TSV files and can support any kind of text data not just typical corpus data annotation layers. CWB requires files to be presented in a single VRT (vertical) file format - this format looks similar to TSV but with no headers and structural data about texts placed all within the same file. Lucene can index various file formats but the simplest way to index is to use text files. This creates a problem for linguistic data marked up with several levels of annotation. To overcome this each of the annotation layers from the original XML was used to generate a text file for that annotation (i.e. one for token, POS, c5 etc.) and then each of these files were indexed as a field within Lucene. This meant although all annotations

[10] https://www.brics.dk/automaton/

[11] https://sourceforge.net/projects/bonnie/

could be queried on, only one could be accessed in a single query.

4.2. Results

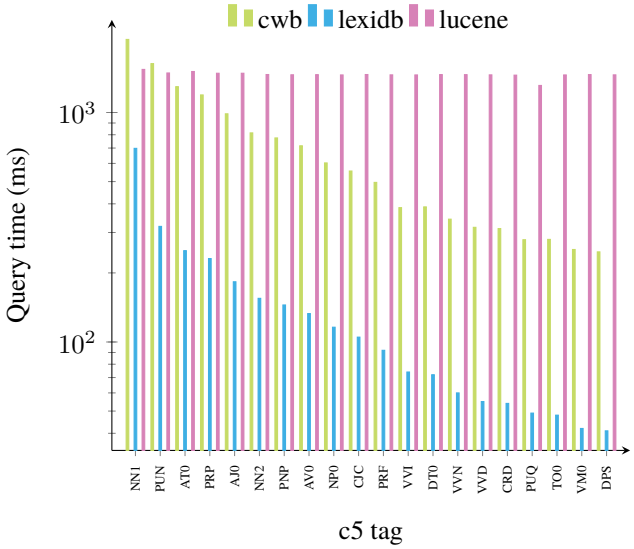

Figure 3: Simple querying for Part-of-Speech (c5 tagset)

As can be seen from figure 3, the LexiDB and CWB query times for searching for single tags (c5) were proportional to the number of occurrences of the tag within the corpus. In this case, 'NN1' being the most popular tag searched for and 'DPS' the least. One would expect this trend to continue with lower frequency items such as low-frequency words or more fine-grained tags (obviously most POS tags will be fairly high frequency within any corpus). LexiDBs query response times in this instance were on average 535% faster than CWB/CQP.

Lucene demonstrated very constant query times for single-term queries (as well as bi and trigrams), but this is due to how Lucene returns a set of hit documents as opposed to retrieving sets of concordances lines. Although Lucene could be used for such linguistic type queries it is necessary to build an additional application layer on top of Lucene to achieve this. This would inevitably end up resulting in greater overhead and slower query response times.

Figure 4 shows the effect of searching for phrases. Using the most popular POS bigrams within the BNC we can see that the results are no longer affected by the raw frequency of the results but rather by the frequencies of the items involved in the lookup. An interesting observation here is that CWB/CQP is slowed more when querying for a phrase where the first item is higher frequency. By comparison, LexiDB is slowed more so when the sum of the frequencies of the items in the phrase is high. We can see that as with unigrams Lucene's response times are fairly constant. This can also be observed when looking at trigrams in figure 5.

Whilst it can be shown from these common POS bigram search results that CWB/CQP is comfortably outperformed by LexiDB which is in turn comfortably outperformed by Lucene, it should be noted that as Lucene is only presenting a set of documents and not a set of hits themselves. CQP and LexiDB are both performing a full query evaluation and retrieval of concordance lines and presenting them to the

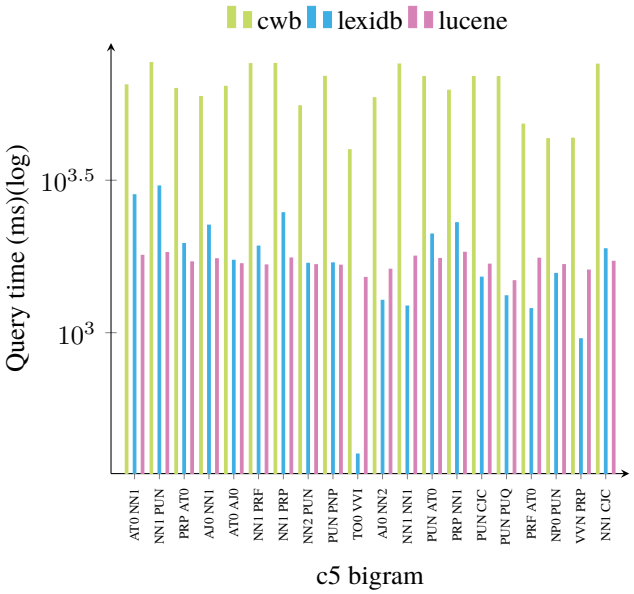

Figure 4: Common POS bigram search

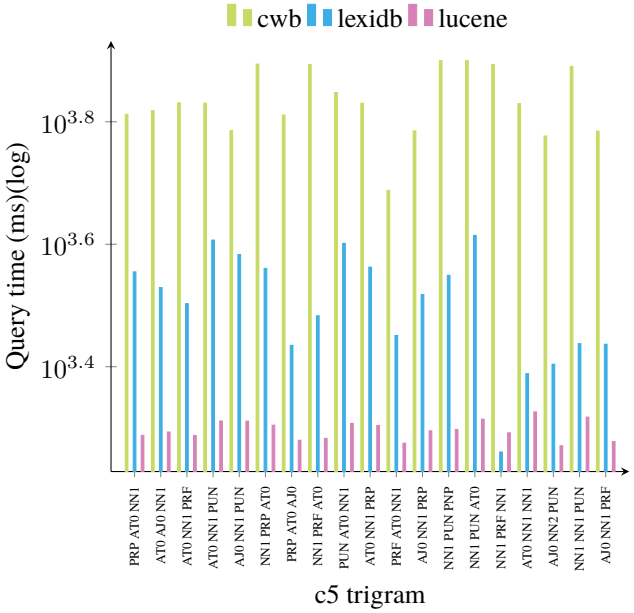

Figure 5: Common POS bigram search

user. Lucene, by contrast, is only performing a lookup of a phrase query and presenting the set of documents the phrase occurs. Additional steps beyond this would need to be taken to present the results to any end-user in a meaningful way. It should also be noted that in this experimental setup Lucene is only capable of searching a single annotation layer at a time. CQP and LexiDB would see similar query response times query across multiple annotation layers. It is worth noting that LexiDB can be scaled out into a cluster setup (as demonstrated in previous papers) to improve performance as needed.

5. Conclusion

In this paper, we have presented two key algorithms critical to the furthering of scalable corpus database management systems. These methods have built upon existing approaches and are highly tailored towards the requirements

of corpus linguistics, digital humanities and lexicography. Our evaluation has shown that the implementation of these methods in LexiDB has proven to be effective in providing better performance than existing CMSs and a level of performance comparable to state-of-the-art indexing systems but with greater capability. Previous work has also demonstrated how the design presented here is scalable and will allow for corpus linguists to utilise ever-larger corpora in the future.

Future work for LexiDB and more generally around the area of corpus databases may include work on examining highly specialised user-interfaces to corpus data systems, both in the form of APIs and graphical user interfaces. This is an area only loosely explored by other systems with highly specific requirements to individual research projects with no general-purpose approaches taken and little consideration for typical HCI methods.

Beyond this exploration of pre-processed n-gram lists as an efficient mechanism for finding and calculating collocation metrics, a typical requirement of corpus linguists, will be explored to examine what trade-off can be made between the additional storage needs of n-grams versus the increased performance compared to calculate collocations on the fly as in the current version of LexiDB.

LexiDB is fully open source[12] and all evaluation scripts[13] used in this paper are freely available.

6. Acknowledgements

This research is funded at Lancaster University by an EPSRC Doctoral Training Grant. Our research has benefitted from discussions within the Corpus Database Project (CDP) team including Laurence Anthony (Waseda University, Japan), Stephen Wattam and John Vidler (Lancaster University, UK).

7. Bibliographical References

Anthony, L. (2004). Antconc: A learner and classroom friendly, multi-platform corpus analysis toolkit. *Proceedings of IWLeL*, pages 7–13.

Baeza-Yates, R. A. and Gonnet, G. H. (1996). Fast text searching for regular expressions or automaton searching on tries. *Journal of the ACM (JACM)*, 43(6):915–936.

Baker, P., Gabrielatos, C., and McEnery, T. (2012). Sketching Muslims: A corpus driven analysis of representations around the word 'Muslim' in the British press 1998–2009. *Applied Linguistics*, 34(3):255–278.

Cafarella, M. J. and Etzioni, O. (2005). A search engine for natural language applications. In Proceedings of the 14th international conference on World Wide Web, pages 442–452. ACM.

Chiarcos, C., Dipper, S., Götze, M., Leser, U., Lüdeling, A., Ritz, J., and Stede, M. (2008). A flexible framework for integrating annotations from different tools and tagsets. *Traitement Automatique des Langues*, 49(2):271–293.

Chubak, P. and Rafiei, D. (2010). Index structures for efficiently searching natural language text. In Proceedings of the 19th ACM international conference on Information and knowledge management, pages 689–698. ACM.

Coole, M., Rayson, P., and Mariani, J. (2015). Scaling out for extreme scale corpus data. In 2015 IEEE International Conference on Big Data, pages 1643–1649. IEEE.

Coole, M., Rayson, P., and Mariani, J. (2016). lexidb: A scalable corpus database management system. In 2016 IEEE International Conference on Big Data (Big Data), pages 3880–3884. IEEE.

Marie-Hélène Corréard, editor. (2002). Lexicography and natural language processing: a Festschrift in honour of BTS Atkins. Euralex.

Davies, M. (2005). The advantage of using relational databases for large corpora: Speed, advanced queries, and unlimited annotation. *International Journal of Corpus Linguistics*, 10(3):307–334.

Diewald, N., Hanl, M., Margaretha, E., Bingel, J., Kupietz, M., Bański, P., and Witt, A. (2016). Korap architecture–diving in the deep sea of corpus data.

Evert, S. and Hardie, A. (2015). Ziggurat: A new data model and indexing format for large annotated text corpora. *Challenges in the Management of Large Corpora (CMLC-3)*, pages 21–27.

Evert, S. (2005). The cqp query language tutorial. *IMS Stuttgart. CWB version*, 2:b90.

Frankel, J. and Pepper, T. (1999). Gnutella. *Web Site-www. gnutella. com*.

Gregory, I. and Paterson, L. (2019). English language and history: Geographical representations of poverty in historical newspapers. In Svenja Adolphs and Dawn Knight (Eds.), *The Routledge Handbook of English Language and Digital Humanities*, Routledge Handbooks in English Language Studies. Routledge, 1.

Hardie, A. (2012). CQPweb – combining power, flexibility and usability in a corpus analysis tool. *International Journal of Corpus Linguistics*, 17(3):380–409.

Jakubíček, M., Kilgarriff, A., Kovář, V., Rychlý, P., and Suchomel, V. (2013). The TenTen corpus family. In 7th Corpus Linguistics Conference, pages 125–127.

Johansson, S., Atwell, E., Garside, R., and Leech, G. (1986). The tagged LOB corpus: User's manual.

Leech, G. (1992). 100 million words of English: the British National Corpus (BNC). *Language Research*, 28(1):1–13.

McEnery, T. and Hardie, A. (2012). Corpus Linguistics: Method, theory and practice. Cambridge University Press.

Meurer, P. (2012). Corpuscle–a new corpus management platform for annotated corpora. *Exploring Newspaper Language: Using the Web to Create and Investigate a Large Corpus of Modern Norwegian*, 49:31.

Møller, A. (2017). dk.brics.automaton – finite-state automata and regular expressions for Java. `http://www.brics.dk/automaton/`.

Rayson, P. (2008). From key words to key semantic domains. *International Journal of Corpus Linguistics*, 13(4):519–549.

Rychlý, P. (2007). Manatee/Bonito - a modular corpus manager. In 1st Workshop on Recent Advances in Slavonic

[12]https://github.com/matthewcoole/lexidb

[13]https://github.com/matthewcoole/lrec_2020_paper

Natural Language Processing, pages 65–70. Masaryk University, Brno.

Schäfer, R. and Bildhauer, F. (2012). Building large corpora from the web using a new efficient tool chain. In Nicoletta Calzolari, et al., editors, LREC, pages 486–493. European Language Resources Association (ELRA).

Schneider, R. (2012). Evaluating DBMS-based access strategies to very large multi-layer corpora. In Challenges in the management of large corpora (CMLC) workshop at Language Resources and Evaluation Conference (LREC). European Language Resources Association (ELRA).

Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, M. F., Dabek, F., and Balakrishnan, H. (2003). Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1):17–32.